The Business Costs of Technical Debt

Calculate the Costs of Technical Debt to Optimize your Software Delivery

February 2021

CodeScene

Abstract

Software organizations face high levels of unplanned work such as bugs and unexpected rework. Since internal software quality lacks visibility, these excess costs tend to manifest themselves as symptoms like long lead times for implementing new features, missed deadlines, and a high pressure on the technical support team.

The lack of visibility also makes it hard to act upon the root cause, or identify the factors to improve; process, team, or the code itself?

In this paper we present an approach to calculate, visualize, and communicate the costs of technical debt and code quality issues. Using these techniques, a technical leader can establish a baseline and set improvement goals that convert into measurable monetary savings and decreased product risks.

The expected monetary returns are significant; as shown in this paper, the typical development organization can increase their feature delivery efficiency by at least 25% by managing technical debt. That translates into getting 25% more developers without additional staffing costs or coordination needs.

Audience: Technical managers, software architects, and development teams.

Key Takeaway: Each section in this paper summarizes the key points. These are:

- Software development is rarely sustainable: **the average organization wastes 23-42% of their development time** due to technical debt.
- Hiring **more developers increases coordination costs**, which in turn makes the development less efficient, particular in codebases rife with technical debt.
- If your organization spends more than 15% on Unplanned Work, then that's a warning sign that delivery potential is wasted. **Technical debt is likely to be a significant chunk of that waste**.
- Technical debt is often mistaken for "bad code in general". This is a dangerous fallacy that **leads organizations to waste months on improvements that don't have a clear business outcome** or aren't urgent.
- Instead, the costs of **technical debt can be quantified** by calculating excess unplanned work via our given formula.
- Based on data, many organizations **pay for 100 developers, but are only getting the output equivalent of 75 developers**.
- Technical debt is only one factor that often comes together with team or process issues that need to be understood and addressed. **Modern tooling helps detecting the bottlenecks.**
- By addressing the root causes, an organization is likely to **increase their effective development size *by least* 25%.**
- 25% extra capacity means you could **deliver more features** and also get a clear win in customer satisfaction due to improved quality.

Technical Debt is a Business Problem

Technical debt is a metaphor where – just like in finance– debt incurs interest payments. This means that technical debt makes our code more expensive to maintain than it has to be.

This has a direct impact on our business. Sustainable software development is about balancing short- and long-term goals. A product needs to grow with new features and capabilities while ensuring that the codebase remains maintainable, easy to evolve, and well-understood.

A failure to balance these goals leads to technical debt. Growing technical debt also puts you at risk for cost overruns and missed commitments. The resulting code quality issues will in turn impact customer satisfaction, with users experiencing bugs and slow innovation. Let's start by looking at some real numbers.

The Costs of Technical Debt

***Key Takeaway**: The average organization wastes 23-42% of their development time due to technical debt.*

A Scandinavian study reveals that developers waste, on average, 23% of their time due technical debt (Besker, T., Martini, A., Bosch, J. (2019) "*Software Developer Productivity Loss Due to Technical Debt*").

As if that wasn't alarming enough, Stripe published data that software developers spend 42% of their work week dealing with technical debt and bad code (Stripe, (2018), "The Developer Coefficient: Software engineering efficiency and its $3 trillion impact on global GDP").

The Talent Link: why hiring more developers isn't the answer

***Key Takeaway**: Hiring more developers leads to increased coordination needs, that in turn can make the development less efficient, particular in codebases rife with technical debt.*

Not only should these numbers be a concern for each business, they also point at a deeper long-term problem: the talent supply. With a 23-42% waste of productivity, companies will have to act to keep their commitments and deliver on their roadmap. A gut response is to hire more people to make up for the wasted capacity.

However, we cannot just hire more developers forever. It's a limited talent pool, and a global competition for skilled developers. And even if we *could* hire as many developers as we'd want, there's evidence that advise against it. Consider the following graph:

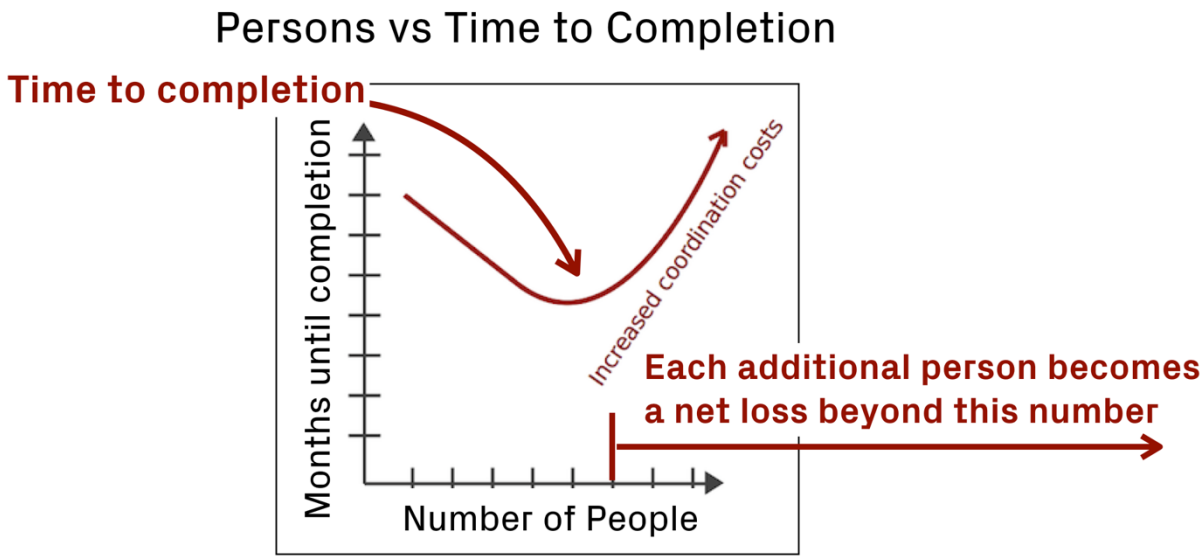


Figure 1: Brooks's Law predicts that adding more people to a late project makes it later.

In essence, the number of available hours grow linearly with each new recruitment. However, the number of possible communication paths between the team members grow much more rapidly and at some point, each additional person becomes a net loss. The gains in hours available is consumed by the additional coordination and communication overhead…and then some (see https://codescene.com/blog/visualize-brooks-law/).

This means that we need to get more work done with fewer people. Possible? Yes, because we have untapped potential. What if we could convert all those hours wasted on technical debt and bad code into productive hours? Let's start by establishing a baseline that we can measure improvements against.

Calculate the Impact of your Technical Debt

Establish a Baseline to Estimate Technical Debt Costs

***Key Takeaway**: If your organization spends more than 15% on Unplanned Work, then that's a warning sign that delivery potential is wasted and technical debt is likely to be a significant chunk of that waste.*

A common barrier to implementing any proposed improvement is the uncertain reward; how much can I save by investing into this? Technical debt management is not different. The main challenge is that most organizations:

a) don't know what their current software quality is,
b) don't know how much their technical debt costs them today, and
c) don't know the business impact of the current quality issues.

Hence, the first step to managing technical debt is to establish a baseline, which gives the organization situational awareness. Before we go there, we need to cover a common mistake: technical debt cannot be calculated from the source code of a system. Let's see why.

Know the Fallacy: Technical Debt cannot be Calculated from Source Code

***Key Takeaway**: Technical debt is often mistaken for "bad code in general". This is a dangerous fallacy that leads organizations to waste months on improvements that don't have a clear business outcome or aren't urgent.*

More specifically, we cannot use traditional static code analysis techniques to estimate or identify technical debt because:

- Technical debt cannot be detected in the source code.
- Technical debt is not equal to code quality issues.
- The cost of technical debt is not the time it would take to refactor the code.

For these reasons, technical debt calculations have to be based on outcome-oriented metrics. Let's see how a measure of unplanned work serves that purpose.

Use Unplanned Work to Calculate ROI

***Key Takeaway**: Based on our data, many organizations pay for 100 developers, but are only getting the output equivalent of 75 developers.*

Unplanned work is any task due to bugs, service interruptions, or flawed software designs. Unplanned work is problematic since it steals capacity and leads to inherently unpredictable delivery that turns an organization into a reactive rather than pro-active entity.

Most organizations track unplanned work indirectly via their product life-cycle management tools like Jira and Azure DevOps. This makes it possible to calculate the ratio of planned vs unplanned work over time:

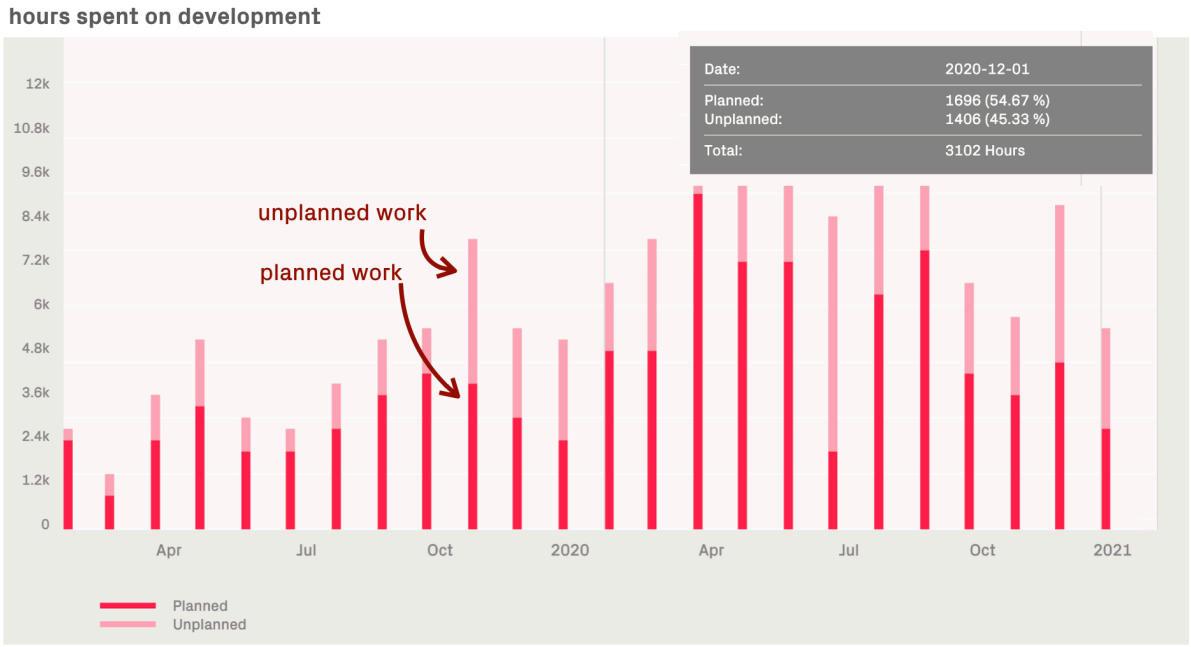


Figure 2 Trend showing the percentage of Unplanned Work over the past year. On average, 40-50% of the development time is wasted on unplanned work.

Once we have a baseline for our development organization, we can calculate a return on investment (ROI) for planned improvements. For that, we need to have a target; what's an acceptable level of unplanned work?

The Formula: Calculate the Untapped Capacity tied up in Technical Debt

We can never eliminate unplanned work entirely. Instead, a good baseline is 15% which is what high-performing organizations achieve (see *Accelerate: The Science of Lean Software and DevOps* (2018) by N. Forsgren PhD, J. Humble, & G. Kim). That 15% baseline lets us establish the following formula:

```
Waste (%) = UnplannedWork% - 0.15
UntappedCapacity ($) = Ndevelopers * AverageSalary * Waste
```

Example: With 40% unplanned work, and 100 developers with an average monthly cost of 7.000k$ we get:

```
Waste (%) = 0.40 - 0.15 = 25%
UntappedCapacity: 100 * 7.000 * 0.25 = 175.000 $ / month
```

Or, put the other way around, you're paying for 100 developers, but get the equivalent of just 75 developers. 25% is wasted. Is that sustainable?

Act: Uncover the Root Causes

***Key Takeaway**: Tech debt is only one factor, and it often comes together with team or process issues that need to be understood and addressed. Modern tooling helps detecting the bottlenecks.*

CodeScene has collected data from several projects across different industries and of different scales. Based on that data, many organizations spend 25-40% of their development capacity on unplanned work, with outliers spending 70-80% on unplanned work.

This span is close to the reported 23-42% waste of productivity due to technical debt (see The Costs of Technical Debt above). However, technical debt is only one factor of several contributing to excess unplanned work. This means we shouldn't handle technical debt in isolation.

More specifically, there are 3 areas that need to be investigated in depth when looking to improve a software delivery flow:

1. Code Quality vs Relevance: not all Technical Debt is Urgent,
2. Ensure Organizational Alignment with the Software Architecture, and
3. Process Loss: Observe the People Side of Code.

The following sections explain each one of these areas.

Code Quality vs Relevance: not all Technical Debt is Urgent

Technical debt often implies severe code quality issues. However, the opposite is not necessarily true: just because some code lacks in quality, that doesn't mean it is or adds to technical debt. It might not even be an immediate problem.

Further, an organization just cannot act on all potential issues at once, so it's vital to prioritize the most critical code quality issues and address them first.

Further reading: Prioritize technical debt based on the development relevance and business impact: https://codescene.com/blog/evaluate-code-quality-at-scale/

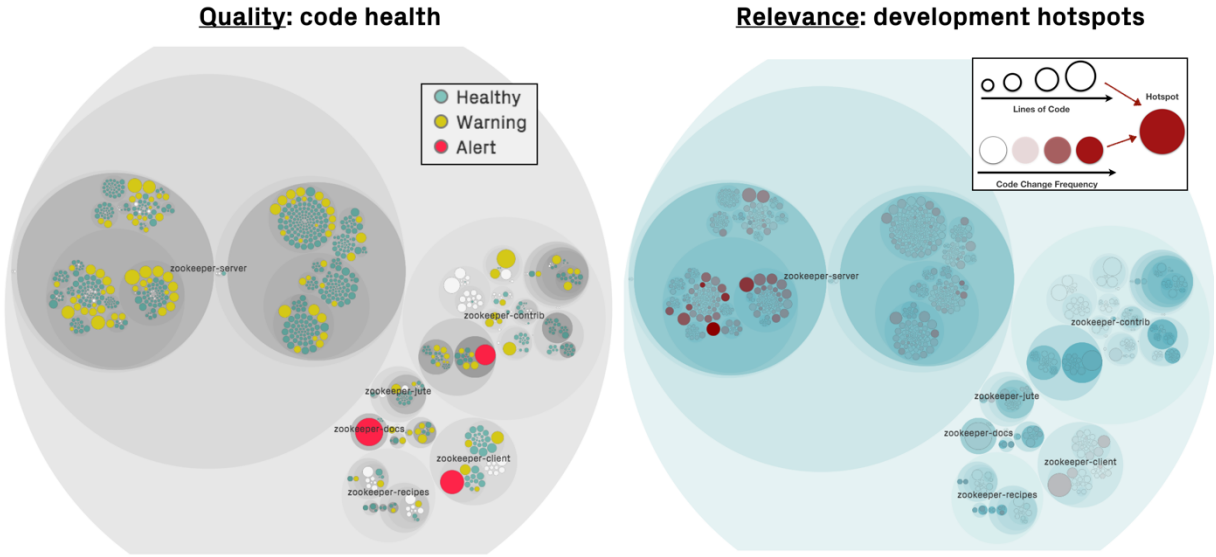


Figure 3 Visualize the code quality in the context of development activity to prioritize and assess the relevance of the findings. Visualizations and measures via https://codescene.io/

Ensure Organizational Alignment with the Software Architecture

A software architecture never exists in a technical vacuum – the architecture must align with the organization (e.g. the development teams). A misaligned architecture leads to increased coordination needs, and an increased risk for defects. Measuring and visualizing your software architecture from the perspective of your development teams is vital.

Resource: Read more on how to visualize logical dependencies across organizational team boundaries: https://codescene.com/blog/codescene-release-3_6/

Process Loss: Observe the People Side of Code

The practices of the development organization can become bottlenecks too. It's important to measure how much time is spent at idle, waiting for some activity to happen. Common examples include long-lived feature branches and Pull Requests awaiting approval. Code can also turn into *knowledge islands*, meaning a component is written mainly by one developer which introduces a key personnel dependency.

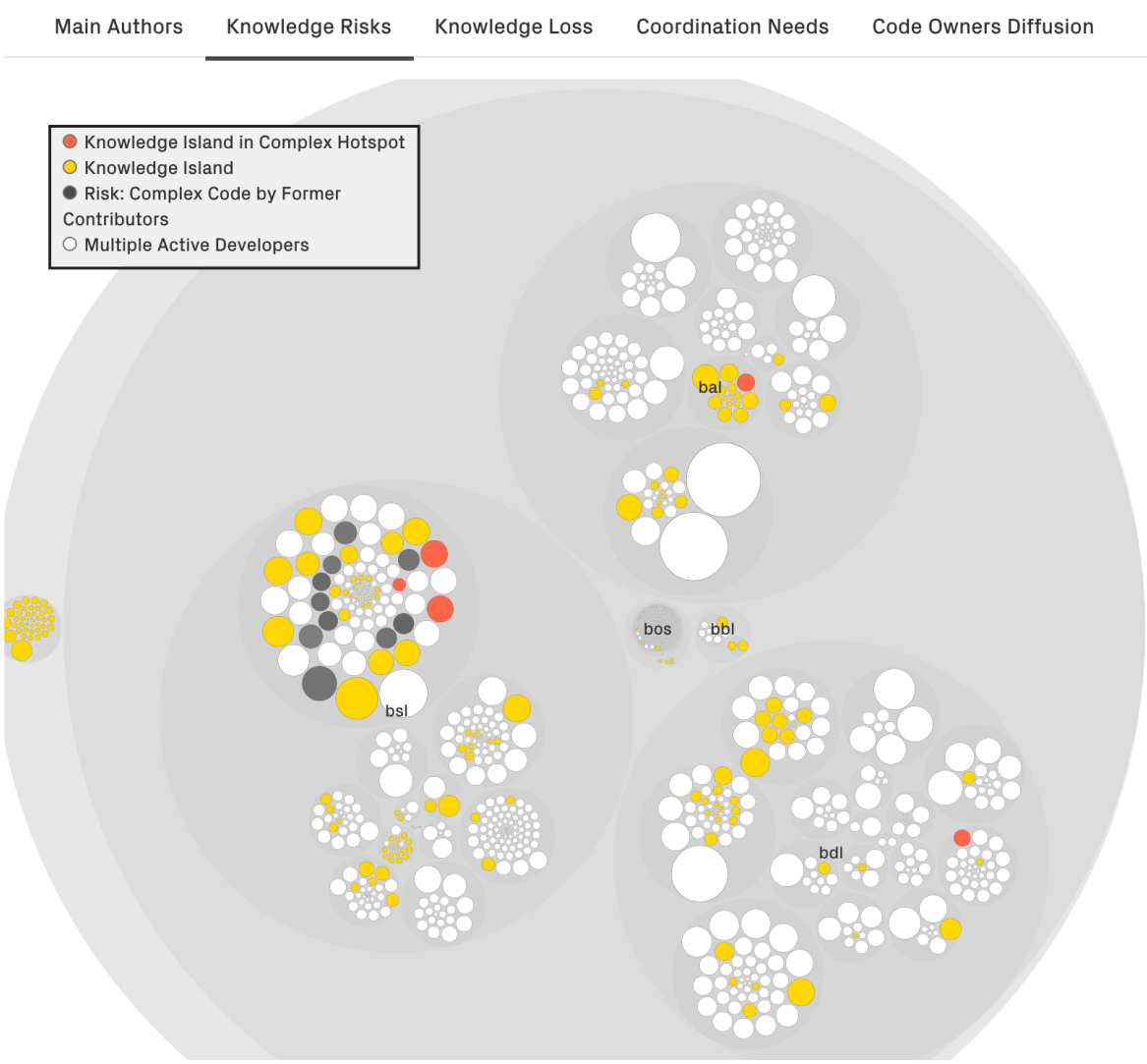


Figure 4 Visualize the knowledge distribution in a codebase. Detect bottlenecks and problematic code with key personnel dependencies. Example via https://codescene.com/

Measure the Expected Outcomes

***Key Takeaway**: The true potential is higher than the savings in unplanned work; the calculations in this paper are on the lower end of the spectrum. Technical debt comes with an opportunity cost too, where planned work takes longer than it should as well.*

Deliver more with the existing Organization

The return on investment when managing technical debt doesn't *have* to translate into cost savings. Rather, it's an opportunity to get more done with the existing organization. If you knew your organizations real potential and could tap into it by paying down technical debt, what would you do with that additional capacity?

- **Time to market**: How much quicker can you iterate on your product roadmap?
- **Motivational boost**: How much more would developer productivity increase just through the motivational boost of no longer have to wade through swaps of technical debt?
- **Quality impact**: How much quicker can you act on reported bugs, and how much will that impact customer satisfaction?

There's more: Technical Debt impacts Planned Work too

Technical debt makes it harder and riskier to modify existing source code. This has consequences for the product and its market. With longer lead times, you just cannot explore all the opportunities you see in your domain. Innovation gets down-prioritized. When translated to a business context, long lead times for new features mean a longer time to market. This makes it hard to keep customer commitments. With technical debt, planned work takes longer too.

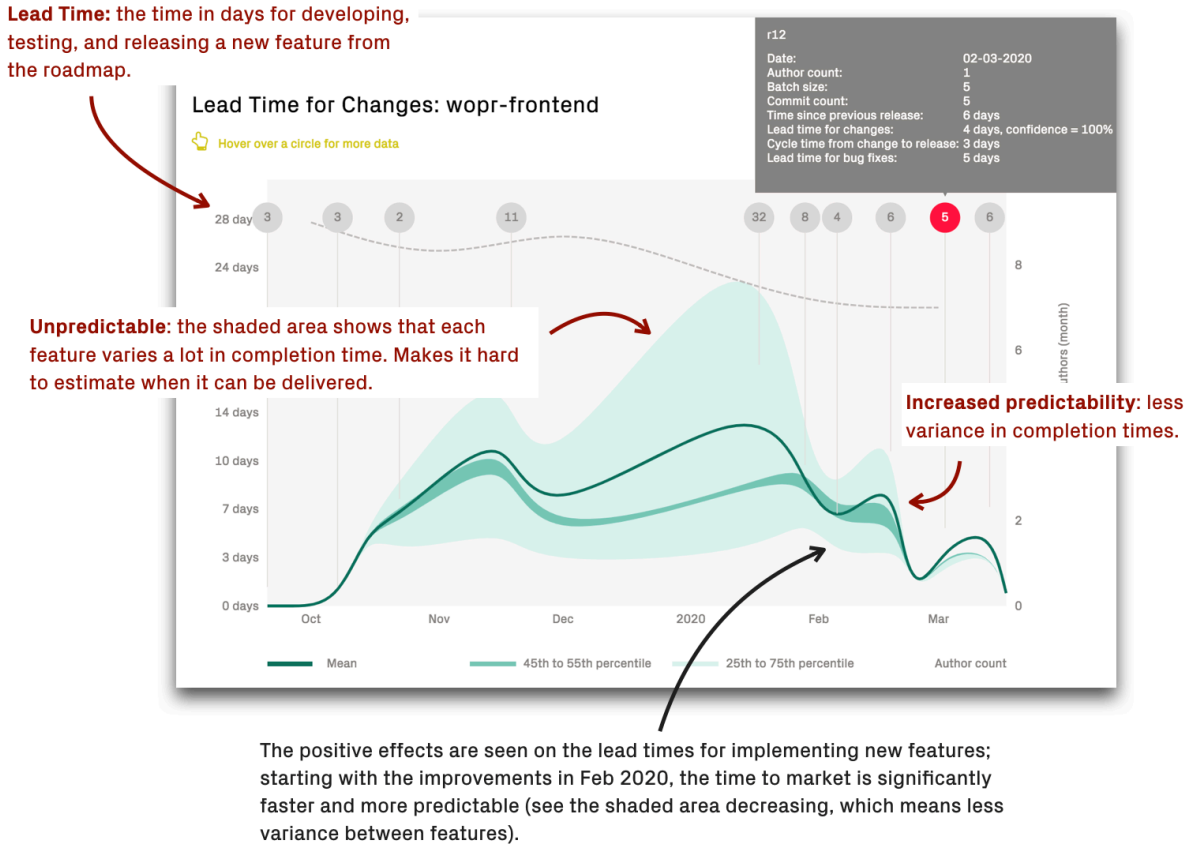


Figure 5 Technical debt impacts the lead time for implementing new product features. [1]

[1] Data and example from https://codescene.com/blog/communicate-technical-improvements-to-non-technical-stakeholders/

Summary

Technical debt is pervasive in the software industry, and accounts for a productivity loss of 23-42%. The key challenges with technical debt are that:

a) Technical debt lacks visibility, making it hard to communicate its cost and impact.

b) Technical debt isn't actually a technical problem, and hence needs to be prioritized and aligned with the business. We cannot "fix" all technical debt at once, neither should we.

c) Adding more people to counter the effects of technical debt doesn't scale well.

This paper presented an approach to estimate the costs of technical debt and its related issues based on the wasted potential of the organization. The calculations are on the lower end of the actual costs; technical debt also impacts planned work, constraints innovation, and effect developer motivation.

Thus, technical debt represents an opportunity; measuring the real potential of your organization gives you a return on investment for managing technical debt in terms of freeing additional capacity to drive your business and product. It's your advantage.

Further Information

The CodeScene Analysis Platform

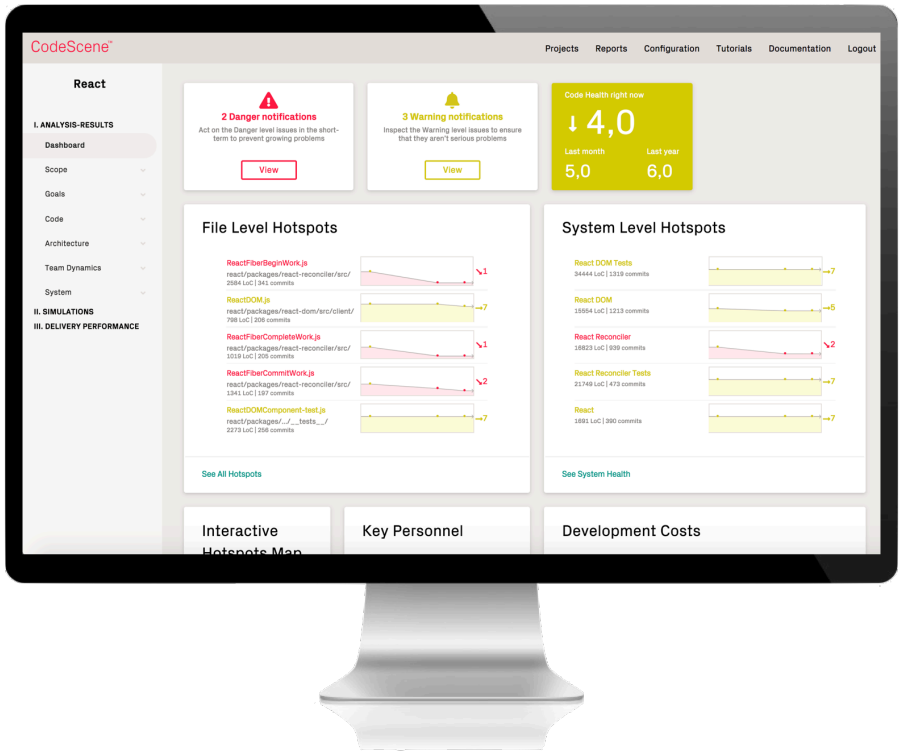


Figure 6 CodeScene prioritizes technical debt measures key organizational metrics.

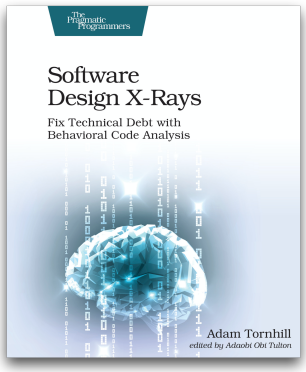
CodeScene is a Swedish startup founded in 2015. CodeScene is a quality visualization platform for software that prioritizes technical debt, detects delivery risks, and measures organizational aspects. Fully automated. The CodeScene platform is used by global Fortune 100 companies in a wide variety of domains.

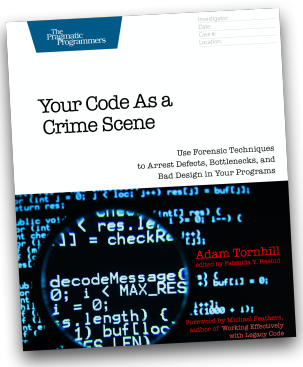
CodeScene automates all the metrics covered in this paper.

Contact: mailto:info@codescene.com
Web page: https://codescene.com/
Twitter: https://twitter.com/codescene
LinkedIn: https://se.linkedin.com/company/codescene

About the Author

Adam Tornhill is a programmer who combines degrees in engineering and psychology. He's the founder of the CodeScene analysis platform. Adam is also the author of Software Design X-Rays, the best-selling Your Code as a Crime Scene, Lisp for the Web and Patterns in C.





Credits

Thanks to Daniel Wellman, Joseph Fahey, Aslam Khan, Peter Caron, and Łukasz Rauer for reading early drafts of this paper. Your feedback and comments made the paper so much better than what I could have done on my own.

I also want to thank all CodeScene users that I had the pleasure of interacting with over the years. This paper grew out of my experience with analyzing real-world codebases and listening to your feedback. Technical debt is a tricky topic, and I want to thank you for all the great conversations where I got to learn about your context and challenges. Thanks!